Distributed Storage Systems for Data Intensive Computing

Sudharshan S. Vazhkudai* Ali R. Butt† Xiaosong Ma‡

Abstract

In this chapter, we will present an overview of the utility of distributed storage systems in supporting modern applications that are increasingly becoming data intensive. Our coverage of distributed storage systems will be based on the requirements imposed by data intensive computing and not a mere summary of storage systems. To this end, we will delve into several aspects of supporting data-intensive analysis, such as data staging, offloading, checkpointing, and end-user access to terabytes of data, and illustrate the use of novel techniques and methodologies for realizing distributed storage systems therein. The data deluge from scientific experiments, observations, and simulations is affecting all of the aforementioned day-to-day operations in data-intensive computing. Modern distributed storage systems employ techniques that can help improve application performance, alleviate I/O bandwidth bottleneck, mask failures, and improve data availability. We will present key guiding principles involved in the construction of such storage systems, associated tradeoffs, design, and architecture, all with an eye toward addressing challenges of data-intensive scientific applications. We will highlight the concepts involved using several case studies of state-of-the-art storage systems that are currently available in the data-intensive computing landscape.

1 Data Intensive Computing Challenges

The advent of extreme-scale computing systems, e.g., Petaflop supercomputers, cyber-infrastructure, e.g., TeraGrid, and experimental facilities such as large-scale particle colliders, are pushing the envelope on dataset sizes. Supercomputing centers routinely generate huge amounts of data, resulting from high-throughput computing jobs. These are often result-datasets or checkpoint snapshots from long-running simulations. For example, the Jaguar petaflop machine [9] at Oak Ridge National Laboratory, which is No. 2 in the Top500 supercomputers as of this writing, is generating terabytes of user data while supporting a wide-spectrum of science applications in Fusion, Astrophysics, Climate and Combustion. Another example is the TeraGrid, which hosts some of NSF's most powerful supercomputers such as Kraken [5] at the University of Tennessee, Ranger [7] at Texas Advanced Supercomputing Center and Blue Waters at National Center for Supercomputing Applications, and are well on their way to produce large amounts of data. Accessing these national user facilities, is a geographically distributed user-base with varied end-user connectivity, resource availability, and application requirements. At the same time, experimentation facilities such as the Large Hadron Collider (LHC) [26] or the Spallation Neutron Source (SNS) [6, 25] will generate petabytes of data. These large datasets are processed by a geographically dispersed user base, often times, on high-end computing systems. Therefore, result output data from High-Performance Computing (HPC) simulations are not the only source that is driving dataset sizes. Input data sizes are growing many fold as well [6, 26, 61, 4].

In addition to these high-end systems, commodity clusters are prevalent and the data they can process is growing manifold. Most universities and organizations host mid-sized clusters, comprising of hundreds of nodes. A distributed user base comes to these machines for a variety of data intensive analyses. In some cases, compute intensive operations are performed at supercomputing sites, while post-processing is conducted at local clusters or high-end workstations at end-user locations. Such a distributed user analysis workflow entails intensive I/O. Consequently, these systems will need to support:

- the staging in of large input data from end-user locations, archives, expermental facilities and other compute centers
- the staging out terabytes of output, intermediate and checkpoint snapshot data to end-user locations or other compute destinations
- the ability to checkpoint terabytes of data at periodic intervals for a long-running computation
- the ability to support high-speed reads to support a running application

In the discussion below, we will highlight these key data intensive operations, the state-of-the-art and the challenges and gaps therein to set the stage for how distributed storage systems can help in optimizing them.

*Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831 USA vazhkudaiss@ornl.gov
†Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061 USA butta@cs.vt.edu
‡Department of Computer Science, North Carolina State University ma@cs.ncsu.edu

Data Staging and Offloading Large input, output and checkpoint data is required to be staged in and out of these systems. With the exponential growth in application input and output data sizes, it is impractical to store all user data indefinitely at HPC centers. Traditionally, centers have operated under the premise that users come to them with all of their storage and computing needs. The legacy of this approach still weighs heavily when it comes to provisioning a center as significant portions of the operational budget is spent on large data stores and archives. End-user data services such as staging and offloading are key I/O operations that can help streamline center scratch space consumption and improve its serviceability.

Timely data offloading is necessary to both protect the job output data from center purge policies [3, 1] as well as to to deliver data on a deadline. This is largely left to the user and is a manual process, wherein users stage out result-data using point-to-point transfer tools such as GridFTP [16], `sftp`, hsi [33], and `scp`. The inherent problem with several point-to-point transfer tools, used to offload data from supercomputers, is that they are only optimized for transfers between two well-endowed sites. For example, the TeraGrid [10] offers several optimizations (TCP buffer tuning, parallel flows, etc.) for GridFTP transfers between the various site pairs that make up the TeraGrid, which are already well connected (10-40 Gbps links). In contrast, data staging and offloading involve providing access to the data to the end-user. How does one move data efficiently from well-provisioned HPC centers to the outside world? More often, users come from smaller universities and organizations with varied connectivity to the HPC center. Thus, efficient and timely staging and offloading of data cannot be ignored as a "last-mile" issue.

The need for such a service is also fueled by the, often, distributed nature of computing services and users' job workflow, which implies that data needs to be shipped to where it is needed. For example, several HPC applications analyze intermediate results of a running job, through visualizations, to study the validity of initial parameters and change them if need be. This process requires the expeditious delivery of the result-data to the end-user visualization application for online feedback. A slightly offline version of this scenario is a pipelined execution, where the output from one computation at supercomputer site A is the input to the next stage in the pipeline, at site B. Large-scale user facilities such as the Spallation Neutron Source (SNS) [6] and Earth System Grid (ESG) [2] that employ distributed workflows are already facing these problems and require efficient data staging and offloading techniques.

The inverse of delivering data to the end-user is to stage the data from a source location to an HPC center. Modern applications usually encompass complex analyses, which can involve staging gigabytes to terabytes of input data, using point-to-point transfer tools (e.g., scp, hsi [27]), from observations or experiments. Many times, the applications also involve comparing the above analyses data against large-scale simulation results to see how theoretical models fit real experimental results. Thus, input data can originate from multiple data sources ranging from end-user sites, remote archives (e.g., HPSS [27]), Internet repositories (e.g., NCBI [49], SDSS [61]), collaborating sites and other clusters that run pieces of the job workflow.

Once submitted, the job waits in a *batch queue* at the HPC center until it is selected for running, while the input data "waits" on the scratch space. HPC centers are heavily crowded and it is not uncommon for a job to spend hours—or even days on end—in the queue. The time a job takes to complete, i.e., *(wall_time + wait_time)*, is also the time the input data spends in the scratch space, in the best case when the data is staged at job submission. In the worst case, which is more common, the data waits longer as users conservatively (manually) stage it in much earlier than job submission, let alone job startup. Thus, there is the need for a timely staging in of job input data so it is able to minimize resource consumption and exposure of data to failure.

From the above usecases, we can state the problem as: *Offload by a specified deadline to avoid being purged*; Or, *Deliver by a specified deadline to ensure continuity in the job workflow*. How can distributed storage systems help address this problem? Solutions in this regard can have a profound impact on data intensive computing.

Checkpointing Checkpointing is an indispensable fault tolerance technique adopted by long-running data intensive applications. These applications periodically write large volumes of snapshot data to persistent storage in an attempt to capture their current state. In the event of a failure, applications recover by rolling-back their execution state to a previously saved checkpoint.

The checkpoint operation and the associated data have unique characteristics. First, applications have distinct phases where they compute and checkpoint; often, these phases occur at regular intervals. Second, checkpointing is a write I/O intensive operation. For instance, consider a 10,000 core job that runs for 12 hours and checkpoints every half hour on a system which has 2 GB of memory. For this job, in the worst case, when all of the memory per core is saved as state information, 500TB of checkpoint data is produced during a run. Such data volumes can overwhelm any storage system. As we scale to petaflop systems, this problem is likely to get acute with the increase in the number of computing cores and the amount of data to be saved at each timestep. Under these conditions, high-resolution checkpointing can easily overwhelm the I/O system. Third, checkpoint data is often written once and read only in case of failure. This suggests that checkpoint images

are seldom accessed beyond the lifetime of an application run or even during the run. Finally, checkpointing, however critical it may be for reliability, is pure overhead from an application standpoint, as time is spent away from useful computation.

Data intensive applications usually deal with checkpointing operations in the following ways. First is node-local storage. It is common practice for jobs running on the individual nodes in a distributed computing environment (e.g., cluster or a desktop grid) to checkpoint to their respective node-local storage. Local storage is dedicated and is not subject to the vagaries of network file system I/O traffic. Moreover, local I/O on even moderately endowed desktops offers around 50-100MB/s. However, local storage is bound to the volatility of the compute node itself. Thus, the locally stored data is lost when the node crashes. Moreover, there is no node-local storage on extreme-scale machines as the the disk is one more component that can fail.

Second is shared file systems. Compute nodes can also checkpoint to a shared, central file server. However, shared file servers are crowded with I/O requests and often have limited space. Shared file servers, accessible to desktop grid-like distributed environments, offer merely tens of MB/s as I/O bandwidth. Clusters usually employ sophisticated SAN storage that are able to offer higher throughput. However, the hundreds of nodes in a cluster, on which processes of a parallel application run, can flood the central server with simultaneous checkpointing I/O operations. In extreme-scale systems–with thousands of processors–the I/O bandwidth bottleneck for checkpointing in writing to central file system (albeit a parallel file system), can be profound. Even though parallel file systems (e.g., Lustre, PVFS, GPFS) offer high I/O throughput (order of tens of GB/s in extreme-scale systems), historically, I/O bandwidth has not scaled with processor frequencies. Further, when the I/O channel is shared across multiple applications, the effective throughput achieved by any given application significantly deteriorates, even in such high-end storage systems. Can novel distributed storage solutions help improve the I/O bandwidth bottleneck seen in checkpointing I/O?

End-User Analysis For most end-users of scientific data, certain stages of their data intensive tasks often require computing, data processing, or visualization at local computers and high-end personal desktop workstations. A local workstation is and will remain an indispensable part of end-to-end scientific workflow environments, for several reasons. First, it provides users with interfaces to view and navigate through data, such as images, timing and profiling data, databases, and documents. Second, users have more control over hardware and software on their personal computers compared to on shared high-end systems (such as a parallel computer), which allows much greater exibility and interactivity in their tasks. Third, personal computers provide convenience in connecting users' computing/visualization tasks with other tools used daily in their work and collaboration, such as editors, spreadsheet tools, web browsers, multimedia players, and visual conference tools. Finally, compared to high-end computing systems that are often built to last for years, desktop workstations at research institutions get updated more often and typically have higher compute power than individual nodes of a large, parallel system. This is especially advantageous for running sequential programs, and there exist many essential scientific computing tools that are not parallel. Applications that were once beyond the capability of a single workstation are now routinely executed on personal desktop computers. The combination of fast CPU and large memory provides scientists with a familiar–yet powerful–computing platform right in their office.

While personal computers are up to their important roles in scientific workflows with advantages in human-computer interface and processing power, storage nowadays usually becomes their limiting factor. Commodity desktop computers are often equipped with limited secondary storage capability and I/O rates. Shared storage in university departments and research labs are mostly provided for hosting ordinary documents such as email and web-pages, and usually comes with small quota, low bandwidth, and heavy workloads. This imbalance between compute power and storage resources leaves scientists with the unattractive choice of remote data access when processing datasets larger than their workstations' available disk space. However, the wide-area network latencies kill performance. How can distributed storage systems help sustain end-user analysis on high-end local workstations?

2 Demands and Requirements for Distributed Storage Systems in Data Intensive Science

Distributed storage is an increasingly important component of end-to-end scientific computing workflows, due to the fact that most of such workflows are inherently distributed themselves. In the majority of cases, data acquisition sites (observatory or experimental instruments), supercomputers or large clusters, and scientific data centers are located outside of scientists' local organization and must be accessed remotely. Data generated by the data collection, experiment, or simulation processes, on the other hand, will not be stored on the data generation site, whose storage facilities are often precious shared resources and only used for transient data storage to help the active tasks running on the facility. Scientific computing users usually have to move their data to their home institute for post-processing, or on archival system for affordable long-term storage, or, as in many cases, both. Bringing data back to their local clusters allows for more efficient data processing, analysis, and visualization, and eventually in most situations scientists view and interact with their data on their office workstations with

display devices. Meanwhile, valuable datasets are often archived on tape systems to prepare for potential, while relatively infrequent, reuse.

While scientific data users have managed to get their job done with basic data movement and management tools, such as FTP, SCP, GridFTP, several factors and trends intensify the need for more powerful distributed storage support and possible paradigm shifts in data storage and movement models. First, there is a growing gap in system sizes between data generation sites (experimentation facilities and supercomputers) and consumption sites (local clusters and office workstations). Without efficient distributed storage infrastructures to aggregate capacities and bandwidths, the use of Peta-scale and Exa-scale computing enabled by cutting-edge supercomputers will be severely limited by scientists' data post-processing storage facilities. Second, the increasing complexity of scientific computing workflows makes it harder and harder to stay with labor-intensive and error-prone manual operations. In addition, such manual operations or scripts are often point-to-point processes that do not easily adapt to changes in computing/storage platforms, nor do they naturally support data sharing or collaboration. Finally, point-to-point data movement and single-site storage will not be able to effectively utilize recent technology advances, such as P2P storage and data distribution, volunteer computing, and cloud computing.

One may wonder whether existing distributed storage and data sharing solutions, mostly developed for commercial or entertainment applications, can be applied to scientific computing. Unfortunately, although the storage, processing, and sharing of scientific data can significantly benefit from many components of existing distributed storage solutions, these processes also possess many unique challenges and requirements that have not been addressed by systems in existence:

- **Level of System integration:** Distributed storage systems for data-intensive science need to locate a balance point between tightly coupled systems that resemble parallel file systems, and loosely coupled systems used in P2P data sharing. On one hand, the separation between storage resources and high-level storage structures is highly desirable to accommodate diverse and ever-changing hardware and software components. On the other hand, data-intensive scientific applications, unlike entertainment data sharing and many other commercial applications, adopt a more tightly coupled computation model, and often demand highly optimized performance.
- **Namespace:** Distributed storage systems need to be able to identify datasets stored using well-defined names. These range from self-identifying uniform resource indices (URI) to simple file names. These are then needed to be organized in some form of a flat or a hierarchical namespace. A flat namespace is easy to implement, but may not scale to large sizes, whereas a hierarchical namespace is flexible but more involved.
- **Granularity:** Typically, media sharing is done at the granularity of entire files. However, the size of datasets involved in modern scientific computing, and the streaming approach adopted in typical workflows entail that the scientific data is handled in small fixed size portions or chunks. The size of chunks can vary from a few KBs to several hundreds of MBs. Consequently, a data transfer architecture designed for scientific computing must efficiently handle varying size chunks, as well as issues of maintaining, finding, and identifying chunks belonging to specific datasets.
- **Resource model:** While dedicated, high-end resources are universally equipped at supercomputing centers, the distributed end-to-end scientific computing workflow provides many practical use cases for contributed (or volunteer) storage, where storage resource owners donate spaces to be aggregated into large, shared storage capacities. This is partly due to that scientific data are often considered less sensitive compared to commercial data, making storage on individually owned and managed devices more acceptable. Also, often resource-constrained, scientists tend to be more open with distributed storage solutions that have low cost, in terms of both hardware purchase and system management.
- **Performance Vs. Space Utilization:** A key design consideration for distributed storage systems is to strike a balance between performance and space tradeoffs. What is the goal of the system? Is it to use a set of distributed resources to provide more storage than what is feasible? Or, is it to bring a set of distributed storage resources to provide faster data access performance? Or, can we achieve a balance between these goals?
- **Reliability:** A distributed storage system needs to be able to store data in a reliable fashion. Since such a storage system can be constructed out of dedicated or commodity components, the reliability semantics has to be robust enough to accommodate any underlying fabric. In any case, recent studies show that the rate of storage system failures is high [60, 51, 62] and that ensuring reliability in large-scale installations is complex. Any distributed storage system will need to support a combination of standard replication and erasure coding schemes depending on space and performance tradeoffs.
- **Transparency:** Transparency is a highly desired feature for data-intensive science. In many situations, transparency translates into ease of use, portability, and reusability that can be of more value than performance. In particular, scientific application developers and users are typically domain scientists, who hesitate to invest time and effort in configuring distributed storage services, or to modify existing applications. In addition, transparent storage solutions allow existing applications and workflows to evolve with new hardware and software upgrades, which is worthwhile

compared to the lost optimization opportunities when more lower-level design and implementation details are exposed to applications and users.

- **Deployability:** Any practical data storage scheme should provide abstractions that can be easily integrated with the application base, and should be minimally intrusive on existing software to ensure adoption by system administrators. Ease of deploying, maintaining, and using a particular service is key to its success as a practical system. For instance, a distributed storage service that uses the standard NFS [20] protocol is more likely to see actual deployment compared to a service which requires users to link with customized libraries, or worse make changes to their code base.
- **Quality of Service:** Quality of service metrics for a distributed storage system range from ensuring that the datasets are safely stored, to ensuring integrity and correctness on retrievals, to securing the datasets against malicious users and hosts and to guaranteeing performance. A loosely coupled, contributory storage poses fundamental challenges to ensuring quality of service.
- **Bulk Data Optimizations:** Distributed storage for data-intensive science has to be designed with handling massive data in mind, in terms of dataset size, access granularity, or both. In particular, with Peta-scale computing centers becoming the main stream, there is a growing disparity between a simulation site and other parts of a scientific computing workflow in storage capacity and bandwidth.
- **Leverage Commodity Components:** Finally, an ideal service will utilize commodity off-the-shelf components for realizing its goals. This is critical, as cost is a major obstacle in large-scale HPC installations, and relying on specialized hardware may make an approach economically non-viable. More and more, there is a wealth of commodity components at end-user sites, in the data path and at the HPC center. Distributed storage systems need to be able to utilize these in a concerted fashion.

The end-to-end data path in scientific computing throws open numerous opportunities to construct novel distributed storage systems that can be brought to bear on I/O intensive tasks. In the following case studies, we will highlight several state-of-the-art distributed storage solutions that are built from novel combinations of storage elements available in the end-to-end I/O stack. We will further analyze how these systems address the data intensive computing challenges.

3 Case Studies in Distributed Storage Systems

3.1 Google File System

The Google File System (GFS) [32] is a distributed file system developed by and deployed at Google, specifically designed to its web data processing and search engine workloads. GFS' design principles are based on Google's data access workload as well as computing platform characteristics.

As Google periodically crawls the web space, downloads web contents, and index es documents to provide continuous and scalable service to many concurrent search engine users, it creates many large files and most of its files are seldom overwritten. Instead, its write workload is heavily made up by appends, where it is common for multiple clients to concurrently append to a shared file. Meanwhile, overall Google has a read-intensive workload, with a large number of current queries processed simultaneously. Several major GFS design decisions reflect these workload requirements. First, files are partitioned into chunks, which are distributed to multiple server nodes, for better access throughput. Second, the chunk size is set at 64MB, much larger than block sizes used in traditional file systems, to reduce the metadata size and communication/management overhead. Third, GFS adopts a relaxed consistency model that targets Google's appending-oriented file mutations. In addition, the general optimization goal of GFS is made to prioritize high throughput over low latency.

Similarly, GFS is highly customized toward Google's computing environment, which consists of large collections of commodity nodes and heavily relies on hardware and software redundancy to protect against failures. GFS' architecture also reflects the same philosophy, where chunk replication plays a key role in both fault tolerance and scalable distributed data accesses. A GFS cluster is made of one master node, multiple chunkserver nodes, and many client nodes. File chunks are aggressively replicated (with a configurable replication degree, which is set at 3 by default). The chunk replicas are intelligently placed to improve data availability and to enhance the network bandwidth utilization. The master nodes manages metadata such as the name spaces, the file-to-chunk mappings, and the chunk locations. Google has demonstrated that with its large chunk size, a single master node is capable of managing and serving large GFS clusters made of thousands of nodes. This has inspired the single-master design in other distributed storage systems such as FreeLoader.

GFS' data storage model and architecture works hand-in-hand with its application interfaces, such as the well-known MapReduce model [29]. With MapReduce, more complex operations can be partitioned into many Map operations that takes input data and generate intermediate results, both in the form of key-value pairs, which are then sorted by the key and passed to nodes that perform the result merging with Reduce tasks. Many of Google's data processing tasks can be

expressed as a pipeline that consists of one or more MapReduce stages. With GFS providing the underlying distributed chunk access services, MapReduce applications can easily perform distributed Map tasks and shuffle data to redistribute intermediate results to reduce tasks. Also the chunk replication mechanism naturally supports the task replication performed by MapReduce for better reliability. Hadoop [34], a popular open-source MapReduce framework implemented by Apache, comes with an open-source counterpart of GFS, called HDFS (Hadoop Distributed File System).

Given Google's read- and append-intensive I/O workload and its loosely coupled distributed execution environments (as opposed to supercomputers or clusters running parallel batch jobs), GFS is suitable for certain classes of scientific data workloads, such as data centers that provides query, mining, and visualization services. On the other hand, though GFS is designed for massive data processing, it is not optimized for highly synchronized, write-intensive applications such as parallel simulations.

3.2 FreeLoader

FreeLoader [70, 71] is a distributed volunteer storage framework developed at North Carolina State University and Oak Ridge National Laboratory, which aggregates unused desktop storage space and I/O bandwidth into a shared cache/scratch space. It was motivated by the observation that even with the the proliferation of high-end systems (high-performance parallel file systems, storage area clusters, data centers, and archival systems), there is a lack of end-to-end storage support for scientists to accommodate, prepare, or consume data in their local computing environments. In particular, the "last mile" in many scientific computing workflows requires data processing and visualization at personal computers, where there are interactive devices as well as more user control on software/tools for viewing and navigating data. While personal computers today are equipped with unprecedented processing power, I/O and storage are more than ever the weakest link in these systems. Therefore, although recent technologies such as the multi-core architecture has brought personal computers the parallel processing capability to enable powerful desktop data processing, the lack of storage space and I/O rates easily prohibits their effective use for data intensive sciences. FreeLoader was proposed to enable these personal computers to pool not only their idle storage spaces, but also under-utilized I/O bandwidths, to create a shared space for scientists to work on their data.

With FreeLoader, workstation owners within a local area network contribute unused disk space, similar to how volunteer computing participants contribute idle CPU cycles using frameworks such as Condor [42] and Entropia [19]. To utilize today's high-speed local area networks for better data access rates, FreeLoader stripes datasets onto multiple participating nodes (called *benefactors*). The aggregate storage space managed by FreeLoader is intended as a cache or scratch space, rather than a general purpose file system or archival system that offer persistent, long-term storage of data. Instead, it targets creating a space much larger than a typical workstation's node-attached secondary storage, to enable scientists to process, analyze, and visualize their "hot" datasets generated by data-intensive experiments or applications. As interest fades on these datasets, they will be replaced by new datasets that are currently of interest to the local FreeLoader users. In addition, such a distributed storage framework would also facilitate data sharing, as colleagues in the same physical organization tend to collaborate and access common datasets [37, 50]. Further, when scientists consume their data, they often work on certain datasets for an extended period of time (typically days or weeks). Considering that data migration from archival systems is limited by transfer rates that are significantly lower than local I/O or LAN throughput [40, 39, 72], as a storage cache FreeLoader exploits data locality to reduce redundant and expensive remote I/O or data migration operations. In a subsequent project [44], the FreeLoader authors also exploited further in this direction by using a local FreeLoader space to only cache *prefixes* of remotely stored datasets to hide the latency in remote data access with a reduced space cost. Such prefix caching is coupled with *collective downloading* to achieve fast data transfer that makes remote data accesses feel like speedy local FreeLoader space operations.

The FreeLoader storage system contends that such a storage model is practical and cost-effective, based on several observations. First, collectively a large amount of disk space remains under-utilized on personal computers within academic or industry organizations. Studies have shown that on average, at least half of the disk space on desktop workstations is idle, and the fraction of idle space increases as the disks become larger [12, 30]. In addition, most workstations are online for the vast majority of the time [22]. Second, disks are cheap today and personal computers are more frequently updated and upgraded compared to higher-end systems. At the same time, off-the-shelf shared storage solutions such as disk arrays and SAN (Storage Area Network) systems are much more expensive and often out of reach for scientists. Therefore, frameworks like FreeLoader allows people to pool distributed storage devices in a reasonably sized organization into a considerably large, yet affordable, shared space. Third, scientific data use patterns have unique characteristics that allow for simplified design, enabling FreeLoader as a user-level, light-weight system. For example, scientific datasets processed at scientists' local environments are often immutable and are safely archived (typically at the mass storage centers co-located with supercomputers or web data repositories [49, 61, 67].). Also, datasets are large and often accessed sequentially. These features

provide FreeLoader with opportunities to focus more on providing a transparent shared storage space and efficiently reading and writing bulk data, rather than traditional distributed storage issues such as data consistency, concurrency control, and reliability.

The FreeLoader architecture comprises contributing benefactor nodes and a management layer that provides services such as data integrity, high performance, load balancing, and impact control. The FreeLoader prototype demonstrated that in addition to the space aggregation benefit, it was able to deliver higher data access rates than traditional storage facilities available in scientists' local computing environments. This is mainly attributed to novel data striping techniques that aggregate a workstation's network communication bandwidth and local I/O bandwidth. The authors also show that security features such as data encryptions and integrity checks can be easily added as filters for interested clients.

Compared to more general-purpose distributed storage systems built on top of contributed devices, such as Farsite [12] and two projects to be discussed later in this chapter (Kosha [17] and TSS [68], FreeLoader is a more specialized system specifically targeting local scientific data processing. Therefore, it does not support full file system functionality, and only implements a very small set of file I/O interfaces to enable Unix-style read/write operations in addition to whole-file operations. On the other hand, it is a very light-weight software cache/scratch space tailored for handling transient uses of bulk scientific data. In addition, the performance impact on the native workload of donor machines is small and can be effectively controlled. Further, we show that Finally, we demonstrate how legacy applications can use the FreeLoader API to store and retrieve datasets. Also, FreeLoader is designed with the capability to dynamically control its resource use to yield to native workloads on storage contributors. This is particularly important as FreeLoader is intended for data-intensive computing in desktop environments, where owners of contributed benefactors also conduct their day-to-day activities. The original FreeLoader development involved performance impact study [70] and a systematic performance impact control mechanism was proposed in a related study [66].

3.3 stdchk

stdchk, a checkpoint storage system, extends the concept of FreeLoader aggregate storage checkpointing operations in HPC applications. Much like how stdin and stdout input/output systems are ubiquitously available to applications, stdchk argues that checkpointing is an I/O intensive operation, requiring a special 'data path'. It ensures that this data path is be made available to HPC applications as a low-cost checkpoint-optimized storage system. stdchk is optimized for the workload: high-speed writes of incremental versions of the same file. stdchk can be used within a desktop grid, where the loosely connected workstation storage is aggregated; it can be used within a cluster where node-local storage can be aggregated; and finally, it can also be used to aggregate memory from processor cores in supercomputers. To this end, stdchk introduces several optimizations to render itself 'checkpoint-friendly' to HPC applications:

- *High write throughput.* stdchk exploits the I/O parallelism that exists inherently in the aggregated storage to provide a suite of write-optimized protocols that enable checkpointing at throughputs higher than what is feasible in current settings.
- *Support for incremental versioning.* stdchk minimizes the size of the data stored using a novel solution to incremental checkpointing that exploits the commonality between successive checkpoint images. Since checkpoint images are chunked and striped in stdchk, it can afford to perform the following optimizations. First is a fixed-size compare-by-hash (FsCH) technique, which divides a file into equal-sized chunks, hashes them and uses the hashes to detect similar chunks. The main weakness of this approach is that it is not resilient to file insertions and deletions. An insertion of only one byte at the beginning of a file prevents this technique from detecting any similarity. Second is content-based compare-by-hash (CbCH). Instead of dividing the file into equal-sized blocks, CbCH detects block boundaries based on content. Compared to FsCH, this approach is more computationally intensive. stdchk experiments have shown that system-level checkpointing can benefit significantly from incremental checkpointing compared to application or library-level checkpointing. A desired side-effect of incremental checkpointing is that it enables applications to checkpoint at a finer granularity.
- *Tunable data availability and durability.* Since stdchk aggregates storage contributions from transient nodes, standard replication techniques are used to ensure data availability and durability. Further, applications can decide the level of data availability/durability they require. The level of redundancy needs to be balanced against overall space availability as that is a finite amount and dictates the serviceability of the storage system. stdchk choose replication against erasure coding for improving the availability of datasets as erasure coding is a compute intensive operations and applications are eager to return to perform useful computation rather than spending more time checkpointing. Consequently, stdchk conducts the replication in the background.

- *Tunable write semantics.* Additionally, stdchk gives applications the ability to choose between a write semantic that is pessimistic (the system call returns only after the desired level of replication is achieved and, consequently, slower) or optimistic (return immediately after data has been written safely once, while replication occurs in the background). This further gives applications control over the write throughput vs. data durability tradeoff.
- *Automatic pruning of checkpoint images.* stdchk offers efficient space management and automatic pruning of checkpoint images. These data management strategies lay the foundation for efficient handling of transient data.
- *Easy integration with applications.* stdchk provides a traditional file system API, using the FUSE (File system in user space) Linux kernel module, for easy integration with applications. Since the entire checkpoint storage is mounted as a file system, applications can save snapshot data seamlessly. This tranparency comes with a small performance cost in the write operations. However, the flexibility offered outweighs this cost.

In extreme-scale systems, where there is no node-local disks, stdchk can be employed by aggregating memory contributions from the user's allocated processor cores. It is common in HPC job submission systems for jobs to oversubscribe for processors to prepare for failure. For example, depending on the failure rate of the machine, a particular job might ask for 12,000 cores instead of the 10,000 cores that it actually needs. The remaining cores are used for failing over processes. stdchk can create an aggregated memory device built out of such pools. This approach has the advantage that it uses the application's own over subscribed processor allocation. However, in such an instantiation, the data striped on to stdchk is drained to a central, stable parallel file system to make room for additional checkpoint data. Thus, stdchk can be used to improve the I/O bandwidth in data intensive applications.

3.4 BADFS

BAD-FS [15] is a distributed file system for handling large, I/O intensive batch workloads on remote computing clusters distributed across the wide area. BAD-FS facilitates staging of data on distributed storage resources, by allowing the users to explicitly specify the data needs of their applications and then factoring the user specifications in data scheduling decisions. BAD-FS differs from traditional distributed file systems in its approach to control data placement and movement. It exposes decisions regarding consistency, caching and replication, commonly hidden inside a file system, to the external scheduler. Using I/O scoping, BAD-FS reduces traffic over the wide area network. Through capacity-aware scheduling, BAD-FS avoids mismatch between jobs and resources, consequently preventing overflowing storage and thrashing caches. The interface exposed by BAD-FS can be leveraged to allow applications to dictate placement of data.

BAD-FS can serve as a enabler for supporting large-scale data staging and offloading. For instance, a user can specify the set of input dataset, locations where the dataset is stored or can be replicated, and locations for storing the output dataset. The scheduler can then stage the data from the specified locations before a job is started, and move the output data to the output locations after completion of the job. Additionally, although not done in BAD-FS, such interfaces can be extended with an automatic monitoring system to allow for dynamic placement of data even in the absence of explicit information from the application.

3.5 dCache

dCache [8] is a distributed storage system to store large datasets that are disseminated from experiments such as the CERN's LHC. It uses a set of commodity nodes to store large datasets and provides access to clients using standard access protocols. Datasets are stored in their entirety on a node and may even be replicated to protect against failure of the commodity node. dCache can be tied to a tertiary storage system and can move data back and forth using LRU schemes. It offers a uniform namespace within a single file system tree for data stored across these storage elements.

Data is usually placed onto pools using pool attraction models that stores data on nodes based on properties such as reliability. Certain pools can be dedicated for interactions with tertiary storage systems. Pools can also communicate between each other to shuffle datasets in order to avoid hot spots in data accesses. Such an approach is used to load balance the dCache storage system.

dCache serves as an excellent use case for storing large data on commodity systems and can help immensely on end-user analysis. Many site have numerous commodity system that can be pooled together to offer a collective storage. However, the I/O throughput offered is limited to the bandwidth capabilities of the individual storage nodes and dCache does not exploit parallelism among the nodes to perform striping.

dCache offers support of grid transfers using the gsiftp [16] mechanims. It also supports the Storage Resource Manager (SRM) [64] protocol. These features make dCache a good candidate for data intensive science and extreme-scale data movement.

There are several similarities with FreeLoader and GFS in how these systems aggregate storage. Contrary to dCache, these system chunk the datasets and stripe them for better throughput. Replication is performed at the chunk-level and not at the

dataset-level as in dCache. While chunk-level operations offer more flexibility, they also entail more management overhead. dCache is fundamentally optimized for providing a large storage space for bulk datasets and accomplishes its goals elegantly.

3.6 IBP

The Internet Backplane Protocol (IBP) [52] is a middleware for managing distributed storage depots. The basic premise behind IBP is to make use of storage in the network fabric. Just like how packets are buffered at intermediate routers on their way from source to destination in the internet, IBP byte arrays are forwarded from one storage depot to another. Therefore, IBP offers a staged approach to data movement, providing application managed communication buffers in the network with a temporal validity. This setup provides a logistical networking infrastructure supporting the scheduling and optimization of data movement for end-to-end applications.

IBP supports the following key functionality:

- Ability to allocate byte arrays for storing data. These allocations can be temporal or permanent; the client can specify whether the allocation is volatile or stable to mean whether the server can revoke the allocatio or not.
- Moving data from senders to byte arrays
- Moving data from byte arrays to receivers

These features are supported using several procedure calls, based on TCP/IP, that help expose a storage to the IBP infrastructure. Distributed storage on a wide-area scale is usually managed and operated using standard file systems with a uniform namespace and strict semantics. Instead, IBP byte arrays can be viewed as files that reside in the network IBP offers applications ways to read and write byte arrays on other depots, thereby creating a shared network resource for storage. IBP byte arrays are append only. IBP offers exNodes to aggregate storage resource across depots to present an aggregate file service over the network. This allows users to interact with IBP infrastructure at a higher-level and not using lower-level services such as storing data in the network. This is similar to users not worrying about disk blocks in file systems

The IBP approach can be used to stage data closer where it is needed or to allow applications to perform their own routing, steering the placement of data in a wide-area setting. Consider the staging in and out of job data between end-users and HPC centers. IBP storage depots can be used as a means to deliver data through the intermediate depots, while also using them as fail-over points in case of resouce failure. Storage depots can be used to move data close to either the end-user or an HPC center. Thus IBP's ability to exploit locality to offer a staged delivery can be used as an alternative to point-to-point transfers in data intensive computing.

IBP's ability to stage data closer to end-user is also similar to FreeLoader's client-side caching. However, IBP is not designed as a locality-aware cache in that users need to explicitly assign temporal validity to files and retention is not based on frequent accesses.

IBP's infrastructure can also be used for distributed checkpointing. As mentioned earlier, checkpoint images are stored on disk within a LAN. However, IBP can be used to store checkpoints in a distributed environment, providing more fault tolerance for snapshot data. This allows end-user applications to control the locations and level of redundancy for checkpoint data. Unlike stdchk, IBP is not specifically geared for checkpointing, but it serves as a nice storage place for checkpoint images.

3.7 Tactical Storage Systems

One key challenge in aggregating distributed (and often heterogeneous) storage hardware for data-intensive scientific applications is to choose a balanced level for I/O interfaces. While systems with a tightly coupled storage hierarchy provides opportunities to deliver highly optimized performance and low overhead, such systems usually lack the portability or flexibility to work with diverse applications/hardware, or to adapt to changes.

TSS (Tactical Storage System) [68] was proposed with the goal of enabling flexible upper-level storage system establishment, by separating storage abstractions from physical storage resources. The TSS authors observed that shared file systems in cluster environments often become major limiting factors in the overall system productivity, in terms of policy constraints, capacity limits, and bandwidth bottlenecks. A TSS allows users to build a variety of storage structures (file systems, databases, or caches), with desired features (distributed and/or shared), on top of storage resources contributed by workstation or cluster owners. TSS was deployed at the University of Notre Dame to support two scientific applications with different storage needs and data use patterns.

Like FreeLoader, TSS operates at user level. Its authors argue that this allows great flexibility in creating different high-level storage abstractions, while the performance disadvantage caused by higher latency and overhead is reasonably small. Its basic storage unit is a file server that exports a Unix-like I/O interface, running on the machine participating in storage aggregation.

However, there are several major distinctions between TSS and FreeLoader. First, TSS is a more loosely coupled and general system compared with FreeLoader. It is intended for building diverse storage abstractions on top of a shared resource layer with well-known and consistent interfaces. FreeLoader, on the other hand, employs an architecture closer to the Google File System, with a single node acting as central manager and meta-data server and participating machines serving chunks of data. Second, TSS aims at flexibility and versatility, therefore its design focus was placed on resource virtualization and abstraction construction (with mechanisms such as *adaptors*, which connect various abstractions to the resource layer). In contrast, FreeLoader is intended to be a shared cache facilitating fast data processing and consumption on desktop workstations, whose design is focused on performance and scalability issues and adopts throughput optimization techniques such as striping. Finally, the TSS prototype has the capability of building a shared file system with Unix-like interfaces, while FreeLoader supports a rather small set of Unix file I/O operations.

3.8 P2P Techniques in Distributed Storage

Peer-to-peer (p2p) overlay networks were initially popularized by file sharing systems such as Napster [48], Gnutella [31], and Kazaa [63]. The main attraction of these systems at the time was their ability to manage a large number of users without any centralized control, and user anonymity that guaranteed freedom from fears of censorship [24]. However, these first-generation systems used centralized servers, proprietary protocols, or controlled flooding for communication among peers in the overlay and for searching data. This led to drawbacks such as bandwidth wastage, lack of resiliency, and dependence on external entities such as *boot* servers. However, studies of p2p traffic on these networks showed their promise as storage substrates: the primary application of these systems was file sharing [41].

The second generation p2p networks imposed some form of structure on the topology of the overlay and formalized the overlay building and maintenance protocols. Examples of such structured p2p overlays include CAN [54], Chord [65], Pastry [56], and Tapestry [75], and have demonstrated the ability to serve as a robust, fault-tolerant, and scalable substrate for a variety of applications [57, 28, 21, 76, 74, 36, 53, 18].

Structured p2p overlay networks essentially implement a *distributed hash table* (DHT) abstraction. Each node in a structured p2p network has a unique node identifier (`nodeId`) and each data item stored in the network has a unique key. The `nodeId`s and keys live in the same name space, and each key is mapped to a unique node in the network. Thus DHTs allow data to be inserted without a-priori knowledge of where it will be stored, and requests for data to be routed without requiring any knowledge of where the corresponding data items are stored, laying the foundation for developing p2p storage systems.

Scalable distributed [35] or serverless [14, 69] file systems provide some p2p aspects. There are also several wide-area file system projects such as Ivy [47], Farsite [13], and Pangaea [58], which also provide reliability.

The basic data sharing is extended by providing strong persistence and reliability in p2p distributed storage projects, such as Pond [55] which is a prototype of Oceanstore [38], CFS [28], and PAST [57].

PAST [57] is a large-scale, Internet-based, storage utility, which uses the p2p network provided by Pastry [56] as a communication substrate. PAST provides scalability, high availability, persistence and security. Any online machine can act as a PAST node by installing the PAST software, and joining the PAST overlay network. A collection of PAST nodes forms a distributed storage facility, and store a file as follows. First, a unique identifier for the file is created by performing a universal hashing function such as SHA-1 [11] on the file name. Next, this unique identifier is used as a key to route a message to a destination node in the underlying Pastry network. The destination node serves as the storage point for the file. Similarly, to locate a file, the unique identifier is created from the file name, and the node on which the file is stored is determined through Pastry routing. PAST utilizes the excellent distribution and network locality properties inherent in Pastry. It also automatically negotiates node failures and node additions. PAST employs replication for fault tolerance, and achieves load-balancing among the participating nodes. Our work builds on the functions provided by PAST to store and retrieve portions of file, and adapts the core PAST functions to handle large files.

CFS [28] provides a scalable, wide-area storage infrastructure for content distribution. CFS exports a file system (hierarchical organization of files) interface to clients. It distributes a file over many servers by chopping every file into small (8 KB) blocks thereby solving the problem of load balancing for the storage and the retrieval of popular big files. This also results in higher download throughput for big files, which can be retrieved in parallel from many nodes. The component that stores data is referred to as a publisher. A publisher identifies a data block by a hash of its contents, and also makes this hash value known for others. Similarly, a client uses the identifier hash of a block and Chord [65] routing to locate and retrieve the block. To ensure authenticity of retrieved data, each block is signed using the publisher's well known public-key. Also, to maintain data integrity, blocks can only be updated by their publishers. Finally, CFS deals with fault tolerance by replicating each data block on k successors, where one successor is made in charge of regenerating new replicas when existing ones fail.

These systems share the goal of using peer nodes to establish a participant-based contributory storage facility, that can be used to support decentralized data delivery and efficient staging in the context of data intensive computing.

Finally, systems such as Kosha [18] and TFS [23] provide transparent access to p2p-storage. In the following, we discuss Kosha in more detail.

3.8.1 Kosha

Kosha [18] provides a Network File System interface [59, 20] to a p2p storage system, and allows users and applications to transparently access their distributed files using a virtual directory hierarchy.

The design of Kosha is aimed at providing storage for individual participating sites consisting of multiple nodes, e.g., clusters connected to the grid. It provides an economical and fault-tolerant alternative to the dedicated storage within a single administrative domain. Kosha instances can provide sustainable intermediate storage locations where data can be stored in a wider end-user data-delivery scheme.

Kosha aims to utilize the cheap storage that is available in targeted environments to create a distributed file system, and to provide features of location transparency, mobility transparency, load balancing, and high availability through file replication and transparent fault handling. These features allow Kosha to run on components that can fail often. For deployability and transparency, Kosha retains the widely used NFS semantics, so that users and applications can access the distributed file system without any changes to their applications.

Kosha organizes the participating nodes into a structured p2p overlay, and uses NFS facilities to make the files available across peers. It ensures that the location of the files remains transparent to the user. Unique to the design of Kosha is that instead of distributing individual files over the distributed storage provided by the nodes in the p2p overlay, it distributes at the level of directories, i.e., files in the same directory are by default stored in the same node as that directory. Kosha also aim to leveraging unused storage space on resources available in academic or corporate settings, where a lot of disk space is wasted on desktop machines.

In Kosha, the participating nodes are assumed to run NFS servers, so that their contributed disk space can be accessed via NFS. It is assumed that only the system administrator has full access to these nodes, and the users cannot modify the system arbitrarily. Various file operations performed are handled as follows. First, Kosha determines the node on which a file is stored by performing a DHT mapping on the file name. Second, the NFS Remote Procedure Calls (RPC) are redirected to appropriate remote nodes. Third, the receiving node performs the operation and returns the results to Kosha, which then records the information needed for future accesses. Finally, Kosha returns control to the client. Hence, the client remains unaware of the underlying RPC forwarding, and the whole operation is transparent, except for a delay caused by the lookup for the appropriate storage node.

By blending the strengths of NFS with those of p2p overlays, Kosha aggregates unused disk space on many computers within an organization into a single, shared file system, while maintaining normal NFS semantics. In addition, Kosha provides location transparency, mobility transparency, load balancing, and high availability through replication and transparent fault handling. Thus, Kosha effectively implements a "Condor" [43] for unused disk storage.

3.8.2 Intermediate Storage Overlays

P2P systems discussed so far utilize loosely connected resources in local or wide area settings to create distributed storage systems. Next, we discuss how a number of such distributed storage sites can facilitate decentralized data delivery, staging and offloading of large data from the perspective of data intensive alaysis within HPC centers.

An issue in using distributed resources is to ensure that data integrity and privacy is preserved during the decentralized transfer. Thus, users often only rely on trusted sites, which are determined using out-of-band agreements. An example of such collaboration can be TeraGrid [10] sites. However, research on decentralized staging [45] and offloading [46] has shown that even when possible participating sites are known a-priori, their dynamic availability and policies entail a discovery process for determining the set of sites that can be used for a particular transfer. Given the scale, dynamic, and distributed nature of intermediate sites, p2p overlays can play a vital role in intermediate site discovery.

P2P Site Discovery The process of selecting intermediate sites (N_i's) from among the participating sites, which are interested in the data transfer, proceeds as follows.

A p2p overlay, e.g., Pastry [56], is used to arrange the N_i's. The overlay provides reliable communication with other participants in the network, even when sites leave or join the system. The participating sites use the overlay to advertise their availability to other nodes in the overlay using random broadcast [46]. Nodes that receive these messages build local information about available nodes for offload. A given node can use its own policies and information about a remote node's capacity to make a decision regarding whether to use the remote node for the offload. For instance, to discover intermediate sites, a user site (N_s) sends out a number of discovery messages on the p2p network with random destination addresses. By virtue of the DHT abstraction provided by p2p routing, the messages are received at some N_i's. On receiving such a discovery message, an N_i replies with its IP address. Thus, N_s *discovers* the N_i. In case the sharing policies of the user site

prohibit it from interacting with N_s, the site can simply ignore the discovery messages from N_s. Finally, to accommodate dynamic preferences of N_i's, N_s discards information about discovered N_i's after a specified period of time and starts a fresh discovery process.

Data-Transfer Paths A decentralized data transfer scheme for HPC centers that ensures timely data delivery and offloading is achieved using a combination of strategies both at the center as well as the end-user to orchestrate the transfers. To this end, the discovered intermediate sites (D_i's) provide multiple data flow paths between the center and the end-user, which lead to better orthogonal bandwidth utilization, faster retrieval speeds, as well as fault-tolerance in the face of failure.

The staging/offloading process works as follows. Let us consider data offloading. Once the job execution completes, the data-offloading process is initiated. First, the center chooses a number of nodes from the set of D_i's ordered by available bandwidth. The exact number of nodes used for this purpose, i.e., the fan-out, is chosen to achieve maximum (pre-specified) out-bound center bandwidth utilization, or to meet previously agreed-upon offload deadlines. These chosen D_i's serve as the Level-1 intermediate nodes. Note that the selected fan-out is not static, and can vary depending on the transfer speeds achieved. Second, the result-data is split into chunks and parallel transfer of the chunks to Level-1 nodes is initiated. Since the Level-1 nodes support better transfer speeds than the user site, the offload time is expected to be much smaller than a direct transfer to the user site. Third, Level-1 intermediate nodes may also further transfer data to the Level-2 intermediate nodes (once again chosen from D_i's), and so on. Consequently, data flows towards the user site, though it is not pushed to the user site. Finally, the user site can asynchronously retrieve the data from the Level-N nodes. Decoupling the user site from the data push path allows the center to offload the data at peak (pre-specified) out-bound bandwidth without worrying about the availability (and connection speed) of the user site, while enabling the user site to pull (retrieve) data from D_i's as necessary.

Similarly, the process of data staging involves the following steps. Once the data staging is initiated, the user site chooses a number of nodes from the set of D_i's (fan-out) ordered by available bandwidth. The cardinality of the fan-out is chosen to stage-in all the necessary data before the predicted job start time. These chosen D_i's serve as the Level-S_1 intermediate nodes. Once again, the selected fan-out is not static, and can vary depending on the actual transfer speeds and the impending deadline. The staging service monitors the changing bandwidths periodically (using NWS [73]) to determine if a chosen fan-out needs to be increased. Next, the input data is split into chunks and parallel transfer of the chunks to Level-S_1 nodes is initiated. The transfer may also involve further levels of intermediate nodes (up to Level-S_N). Alternatively, depending on the availability of intermediate nodes, the user site can also stage the data to Level-S_N nodes much earlier than the deadline.

As the job startup deadline approaches, the close proximity of the Level-S_N nodes to the center allows them to quickly move the input data to the center's scratch space. Also, this design allows the Level-S_N nodes to stage the data at peak (pre-specified) bandwidth at the most appropriate time without worrying about the availability (and connection speed) of the submission site.

The use of intermediate nodes in the decentralized data transfer systems provide multiple data-flow paths between the center and the user site, leading to several alternative options for data delivery. For instance, data may be replicated across different D_i's during the transfer from one level to the other. This will allow for pulling data from a number of locations when needed, thus providing fault tolerance against node failure, as well as better utilization of the available orthogonal bandwidth. Finally, the schedule can also be used to simultaneously deliver data to multiple interested sites in the network.

The use of intermediate nodes is similar to that of IBP. IBP offers a data distribution infrastructure with a set of strategically placed resources (storage deopts) to move data, and implement what is referred to as logistical netwroking. The intermediate storage overlay also exploits the presence of pre-installed storage nodes for data delivery as and when they are available. However, it differs significantly in its attempt to combine both a staged as well as a decentralized data delivery. The induction of user-specified nodes also allows the system to optimize the data delivery on a per user basis, which is not possible with IBP. Further, it strives to meet a deadline in delivering as well as in timely offloading from the HPC center.

References

[1] UC/ANL Teragrid Guide. http://www.uc.teragrid.org/tg-docs/user-guide.html#disk, 2004.
[2] Earth system grid. http://www.earthsystemgrid.org, 2006.
[3] NCCS.GOV File Systems. http://info.nccs.gov/computing-resources/jaguar/file-systems, 2007.
[4] Laser Interferometer Gravitational-Wave Observatory. http://www.ligo.caltech.edu/, 2008.
[5] National Institute of Computational Sciences. http://www.nics.tennessee.edu/computing-resources/kraken, 2008.
[6] Spallation Neutron Source. http://www.sns.gov/, 2008.
[7] Sun constellation linux cluster. http://www.tacc.utexas.edu/resources/hpcsystems/#constellation, 2008.
[8] dCache.ORG. http://www.dcache.org/, 2009.
[9] National Center for Computational Sciences. http://www.nccs.gov/, 2009.
[10] Nsf teragrid. http://www.teragrid.org, 2009.

[11] F. 180-1. Secure Hash Standard. Technical Report Publication 180-1, Federal Information Processing Standard (FIPS), NIST, US Department of Commerce, Washington D.C., Apr. 1995.
[12] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J.Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.
[13] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proc. 5th USENIX OSDI*, pages 1–14, Boston, MA, Dec. 2002.
[14] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, 1996.
[15] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit control in a batch-aware distributed file system. In *Proc. 1st USENIX NSDI*, pages 365–378, San Francisco, CA, Mar. 2004.
[16] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A data movement and access service for wide area computing systems. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, 1999.
[17] A. Butt, T. Johnson, Y. Zheng, and Y. Hu. Kosha: A peer-to-peer enhancement for the network file system. In *Proceedings of Supercomputing*, 2004.
[18] A. R. Butt, T. A. Johnson, Y. Zheng, and Y. C. Hu. Kosha: A peer-to-peer enhancement for the network file system. *Journal of Grid Computing: Special issue on Global and Peer-to-Peer Computing*, 4(3):323–341, 2006.
[19] B. Calder, A. Chien, J. Wang, and D. Yang. The Entropia virtual machine for desktop grids. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, 2005.
[20] B. Callaghan. *NFS Illustrated*. Addison-Wesley Longman, Inc., Essex, UK, 2000.
[21] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralised application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC) (Special issue on Network Support for Multicast Communications)*, 20(8):100–110, 2002.
[22] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 63(5), 2003.
[23] J. Cipar, M. D. Corner, and E. D. Berger. TFS: A transparent file system for contributory storage. In *Proc. 5th USENIX FAST*, pages 215–229, San Jose, CA, Feb. 2007.
[24] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System, 1999. `http://freenetproject.org/freenet.pdf`.
[25] J. W. Cobb, A. Geist, J. A. Kohl, S. D. Miller, P. F. Peterson, G. G. Pike, M. A. Reuter, T. Swain, S. S. Vazhkudai, and N. N. Vijayakumar. The neutron science teragrid gateway: a teragrid science gateway to support the spallation neutron source: Research articles. *Concurrency and Computation : Practice and Experience.*, 19(6):809–826, 2007.
[26] Conseil Européen pour la Recherche Nucléaire (CERN). LHC– the large hadron collider, July 2007. `http://lhc.web.cern.ch/lhc/`.
[27] R. Coyne and R. Watson. The parallel i/o architecture of the high-performance storage system (hpss). In *Proceedings of the IEEE MSS Symposium*, 1995.
[28] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. SOSP*, pages 202–215, Banff, Alberta, Canada, Oct. 2001.
[29] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, 2004.
[30] J. Douceur and W. Bolosky. A large-scale study of file-system contents. In *Proceedings of SIGMETRICS*, 1999.
[31] J. Frankel and T. Pepper. The Gnutella protocol specification v0.4, 2003. `http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf`.
[32] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Proceedings of the 19th Symposium on Operating Systems Principles*, 2003.
[33] M. Gleicher. HSI: Hierarchical storage interface for HPSS. http://www.hpss-collaboration.org/hpss/HSI/.
[34] Hadoop. http://hadoop.apache.org/core/.
[35] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.
[36] Y. C. Hu, S. M. Das, and H. Pucha. Exploiting the Synergy between Peer-to-Peer and Mobile Ad Hoc Networks. In *Proc. HotOS IX*, May 2003.
[37] A. Iamnitchi, M. Ripeanu, and I. Foster. Small-world file-sharing communities. In *Infocom*, 2004.
[38] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent store. In *Proc. ASPLOS*, pages 190–201, Cambridge, MA, Nov. 2000.
[39] J. Lee, X. Ma, R. Ross, R. Thakur, and M. Winslett. RFS: Efficient and flexible remote file access for MPI-IO. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2004.
[40] J. Lee, X. Ma, M. Winslett, and S. Yu. Active buffering plus compressed migration: An integrated solution to parallel simulations' data transport needs. In *Proceedings of the 16th ACM International Conference on Supercomputing*, 2002.
[41] N. Leibowitz, A. Bergman, R. Ben-Shaul, and A. Shavit. Are file swapping networks cacheable? Characterizing p2p traffic. In *Proc. 7th International Workshop on Web Content Caching and Distribution (WCW7)*, Boulder, CO, August 2002.
[42] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, 1988.
[43] M. J. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A hunter of idle workstations. In *Proc. ICDCS*, pages 104–111, San Jose,

CA, June 1988.
[44] X. Ma, S. Vazhkudai, V. Freeh, T. Simon, T. Yang, and S. L. Scott. Coupling prefix caching and collective downloads for remote data access. In *Proceedings of the ACM International Conference on Supercomputing*, 2006.
[45] H. Monti, A. R. Butt, and S. S. Vazhkudai. Just-in-time staging of large input data for supercomputing jobs. In *Proc. ACM Petascale Data Storage Workshop*, Austin, TX, Nov. 2008.
[46] H. Monti, A. R. Butt, and S. S. Vazhkudai. Timely offloading of result-data in hpc centers. In *Proc. 22nd ACM International Conference on Supercomputing (ICS'08)*, Kos, Greece, Jun. 2008.
[47] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. 5th USENIX OSDI*, pages 31–34, Boston, MA, Dec. 2002.
[48] Napster. http://www.napster.com/.
[49] National center for biotechnology information. http://www.ncbi.nlm.nih.gov/.
[50] E. J. Otoo, D. Rotem, and A. Romosan. Optimal file-bundle caching algorithms for data-grids. In *Proceedings of Supercomputing*, 2004.
[51] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proc. USENIX FAST*. USENIX Association, 2007.
[52] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the network. In *Proceedings of the Network Storage Symposium*, 1999.
[53] H. Pucha, S. M. Das, and Y. C. Hu. Imposing route reuse in mobile ad hoc network routing protocols using structured peer-to-peer overlay routing. *IEEE Transactions on Parallel and Distributed Systems*, 17(12):1452–1467, 2006.
[54] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *Proc. SIGCOMM*, San Diego, CA, Aug. 2001.
[55] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The Oceanstore prototype. In *Proc. 2nd USENIX FAST*, pages 1–14, San Francisco, CA, Dec. 2003.
[56] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, pages 329–350, Heidelberg, Germany, Nov. 2001.
[57] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. SOSP*, pages 188–201, Chateau Lake Louise, Banff, Canada, Oct. 2001.
[58] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proc. 5th USENIX OSDI*, pages 15–30, Boston, MA, Dec. 2002.
[59] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proc. Summer USENIX*, pages 119–130, Portland, OR, June 1985.
[60] B. Schroeder and G. A. Gibson. Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you? In *Proc. USENIX FAST*, 2007.
[61] Sloan digital sky survey. http://www.sdss.org, 2005.
[62] S. Shah and J. Elerath. Reliability analysis of disk drive failure mechanisms. *RAMS*, 2005.
[63] Sharman Networks. Kazaa Media Desktop, 2004. `http://www.kazaa.com/index.htm`.
[64] A. Shoshani, A. Sim, and J. Gu. Storage resource managers: Essential components for the grid. In J. Nabrzyski, J. Schopf, and J. Weglarz, editors, *Grid Resource Management: State of the Art and Future Trends*, 2003.
[65] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. SIGCOMM*, San Diego, CA, Aug. 2001.
[66] J. Strickland, V. Freeh, X. Ma, and S. Vazhkudai. Governor: Autonomic throttling for aggressive idle resource scavenging. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing*, 2005.
[67] A. Szalay and J. Gray. The world-wide telescope. *Science*, 293(14):2037–2040, 2001.
[68] D. Thain, S. Klous, J. Wozniak, P. Brenner, A. Striegel, and J. Izaguirre. Separating abstractions from resources in a tactical storage system. In *Proceedings of Supercomputing*, 2005.
[69] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proc. SOSP*, pages 224–237, Saint-Malo, France, Oct. 1997.
[70] S. Vazhkudai, X. Ma, V. Freeh, J. Strickland, N. Tammineedi, and S. Scott. Freeloader: Scavenging desktop storage resources for bulk, transient data. In *Proceedings of Supercomputing*, 2005.
[71] S. Vazhkudai, X. Ma, V. Freeh, J. Strickland, N. Tammineedi, T. Simon, and S. Scott. Constructing collaborative desktop storage caches for large scientific datasets. *ACM Transactions on Storage (TOS)*, 2(3):221–254, 2006.
[72] S. Vazhkudai, J. Schopf, and I. Foster. Predicting the performance of wide-area data transfers. In *Proceedings of the 16th Int'l Parallel and Distributed Processing Symposium (IPDPS 2002)*, 2002.
[73] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computing Systems*, 15(5):757–768, 1999.
[74] R. Zhang and Y. C. Hu. Borg: A hybrid protocol for scalable application-level multicast in peer-to-peer networks. In *Proc. 13th NOSSDAV Workshop*, June 2003.
[75] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-Resilient Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, Apr. 2001.
[76] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. In *Proc. 11th NOSSDAV Workshop*, June 2001.